The Programmer's Guide to PBRT

Chapter 3: Geometry and Transformations

Gemini (AI Thought Partner)

November 29, 2025

Contents

1 Introduction

Welcome to the mathematical foundation of Physically Based Rendering (PBRT). As a programmer with a background in Pre-Calculus, you are well-equipped to handle this chapter.

Good News: Chapter 3 does *not* require Calculus. There are no integrals or derivatives here. Instead, this chapter relies entirely on **Linear Algebra** and **Trigonometry**.

In computer graphics, we do not simply "draw" shapes; we simulate light interacting with matter in a 3D environment. To do this, we need rigorous data structures to represent where things are (*Points*), where they are going (*Vectors*), and how they are oriented (*Normals*).

This document interprets the mathematical concepts of PBRT Chapter 3 through the lens of a C++ programmer, focusing on data types, valid operations, and geometric intuition.

2 Coordinate Systems

Before placing any object in a scene, we must define the "stage" upon which it sits. A coordinate system is defined by an origin point $(0, 0, 0)$ and three basis vectors representing the axes: x, y, z.

2.1 Handedness: Left vs. Right

This is the most common source of bugs in graphics programming. PBRT uses a **Left-Handed Coordinate System**.

- **Right-Handed (Standard Math/OpenGL):** If x points right and y points up, the z-axis points *towards* you.
- **Left-Handed (PBRT/DirectX):** If x points right and y points up, the z-axis points *away* from you (into the screen).

Critical Warning

If you assume the wrong handedness, your cross products (which calculate perpendicular vectors) will point in the opposite direction. This effectively flips your geometry inside out or causes lighting calculations to fail because surfaces will face the wrong way.

3 The Actors: Vectors, Points, and Normals

In standard mathematics, the tuple (x, y, z) is often generically called a "vector." In PBRT, we strictly distinguish between three types to prevent logical errors. As a programmer, think of this as **Strong Typing**.

3.1 Vectors (`Vector3f`)

A vector represents a **direction** and a **magnitude** (length). It does *not* have a position.

- **Analogy:** An instruction like "Move 5 meters North." This instruction is valid whether you are in New York or Tokyo.
- **Data Structure:** ‘float x, y, z;‘
- **Valid Operations:**

$$\mathbf{v} + \mathbf{v} = \mathbf{v} \quad \text{(Go North, then Go East = Go Northeast)} \tag{1}$$

$$s \cdot \mathbf{v} = \mathbf{v} \quad \text{(Scalar multiplication scales the length)} \tag{2}$$

3.2 Points (`Point3f`)

A point represents a specific **location** in space relative to the origin.

- **Analogy:** A GPS coordinate or street address.
- **Data Structure:** 'float x, y, z;'
- **Valid Operations:**

$$\text{Point} + \text{Vector} = \text{Point} \quad \text{(Start at home, walk 5 miles} \rightarrow \text{New Location)} \tag{3}$$

$$\text{Point} - \text{Point} = \text{Vector} \quad \text{(The difference between two locations is a distance/direction)} \tag{4}$$

- **Invalid Operation:** Point + Point. (Adding two street addresses together creates nonsense).

3.3 Normals (`Normal3f`)

A Normal is a vector that is strictly **perpendicular** to a surface. It defines orientation.

- **Why a separate class?** Although mathematically similar to vectors, Normals behave differently under transformations (scaling). If you flatten a sphere into a pancake, the surface vectors flatten, but the Normals must point *more vertically* to remain perpendicular to the surface.
- **Transformation Rule:** Normals are transformed by the **Inverse Transpose** of the transformation matrix (M^{-T}), whereas Vectors are transformed by M.

4 The Toolset: Dot and Cross Products

You do not need to memorize the arithmetic implementation, but you must understand the *geometric intent.*

4.1 The Dot Product

The dot product of two vectors $\mathbf{v}$ and $\mathbf{w}$ is a scalar value (a single number).

$$\mathbf{v} \cdot \mathbf{w} = x_v x_w + y_v y_w + z_v z_w = \|\mathbf{v}\|\|\mathbf{w}\| \cos\theta \tag{5}$$

Programmer's Intuition: This acts as a "Similarity Check."

- **1.0:** Vectors are perfectly aligned (normalized).
- **0.0:** Vectors are perpendicular (orthogonal).
- **-1.0:** Vectors point in opposite directions.

Usage in PBRT: Calculating lighting. If a light ray hits a surface, the intensity is determined by the angle between the light direction and the surface Normal (Lambert's Cosine Law).

4.2 The Cross Product

The cross product of two vectors **v** and **w** returns a new **Vector** that is perpendicular to both inputs.

$$\mathbf{v} \times \mathbf{w} = \begin{pmatrix} y_v z_w - z_v y_w \\ z_v x_w - x_v z_w \\ x_v y_w - y_v x_w \end{pmatrix} \tag{6}$$

Usage in PBRT: Constructing coordinate systems. If you have two edges of a triangle, the cross product gives you the surface Normal. *Note: Order matters!* $\mathbf{v} \times \mathbf{w} = -(\mathbf{w} \times \mathbf{v})$.

5 Rays (`Ray`)

In ray tracing, the Ray is the fundamental probe we shoot into the scene. It is a semi-infinite line defined by an origin and a direction.

5.1 The Ray Equation

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{7}$$

Where:

- **o**: The **Origin** (Point).
- **d**: The **Direction** (Vector).
- t: The **Parameter** (Scalar distance).

5.2 Implementation Detail: `tMax`

The PBRT 'Ray' class contains a mutable member 'tMax'.

- Initially, 'tMax' is set to Infinity.
- When the ray intersects an object at distance t_{hit}, if $t_{hit} < tMax$, we record the hit and update 'tMax = t_{hit}'. $This ensures that subsequent intersection checks ignore objects behind the closest wall we've found so$

6 Bounding Boxes (`Bounds3f`)

Checking for intersection against millions of triangles is computationally expensive. To optimize this, we wrap complex objects in Axis-Aligned Bounding Boxes (AABBs).

- **Definition:** Defined by two points: p_{min} (minimum x,y,z) and p_{max} (maximum x,y,z).
- **Union:** The union of two boxes is a new box that encompasses both.
- **Intersection:** We check if a Ray passes through the box using the "Slab Method." If the ray misses the box, we skip all geometry inside it.

7 Transformations

This is the core mathematical complexity of Chapter 3. We use matrices to move (translate), rotate, and scale objects.

7.1 Homogeneous Coordinates

A 3x3 matrix can rotate and scale, but it cannot translate (move) the origin. To solve this, we increase our dimension to 4D. This is called **Homogeneous Coordinates**.

We add a 4th component, w:

- **Point:** $(x, y, z, 1)$. The 1 indicates this is a location that can be translated.
- **Vector:** $(x, y, z, 0)$. The 0 indicates this is a direction. Adding a translation value to a direction is meaningless (Moving "North" 5 meters to the right is still "North").

7.2 The Transformation Matrix

PBRT uses 4×4 matrices.

$$M = \begin{pmatrix} m_{00} & m_{01} & m_{02} & t_x \\ m_{10} & m_{11} & m_{12} & t_y \\ m_{20} & m_{21} & m_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8}$$

When we multiply a point by this matrix, the t_x, t_y, t_z column is added to the point's position. When we multiply a vector (where $w = 0$), that column is multiplied by 0 and ignored.

7.3 The `Transform` Class

In C++, the 'Transform' class maintains two matrices:

```
class Transform {
  private:
    Matrix4x4 m, mInv;
  // ...
};
```

We store the inverse matrix ('mInv') because we frequently need to transform rays from World Space back into Object Space to perform intersections efficiently.

8 Spherical Geometry

Lights and cameras operate using angles rather than Cartesian coordinates.

8.1 Solid Angle

In 2D, we measure an angle in radians (length of arc on a unit circle). In 3D, we measure a **Solid Angle** ($d\omega$) in **Steradians**. It represents the area of a patch on a unit sphere.

$$d\omega = \sin\theta \, d\theta \, d\phi \tag{9}$$

This formula accounts for the fact that latitude bands on a sphere get smaller as you get closer to the poles (the singularities).

9 Interactions (`Interaction`)

PBRT consolidates all geometric data generated by a ray hitting a surface into a struct called 'Interaction'.

When a ray hits a primitive, it populates this struct with:

- **p**: The Point of intersection.

- $\mathbf{n}$: The geometric Normal at that point.
- $\mathbf{w_o}$: The outgoing direction (vector towards the camera).
- (u, v): Texture coordinates.

This 'Interaction' object is passed to shading functions to calculate color/material appearance.

10 Summary for the Programmer

You have now surveyed the "Linear Algebra Toolkit" required for ray tracing.

1. **Type Safety:** Never mix Points and Vectors indiscriminately.
2. **Coordinate Systems:** Always remember PBRT is Left-Handed ($+Z$ is forward).
3. **Normals:** Require special handling (M^{-T}) during transformation.
4. **Homogeneous Coordinates:** The "magic" w component handles the distinction between translation-sensitive Points and translation-invariant Vectors.

With these structures defined, Chapter 4 will move on to Radiometry (the physics of light measurement).